Main Memory: Address Translation

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Can’t We All Just Get Along?

Physical Reality: different processes/threads share the same hardware → need to multiplex

- CPU (temporal)
- Memory (spatial)
- Disk and devices (later)

Why worry about memory sharing?

- Complete working state of process and/or kernel is defined by its data (memory, registers, disk)
- Don’t want different processes to have access to each other’s memory (protection)

Aspects of Memory Multiplexing

Isolation

Don't want distinct process states collided in physical memory (unintended overlap → chaos)

Sharing

Want option to overlap when desired (for efficiency and communication)

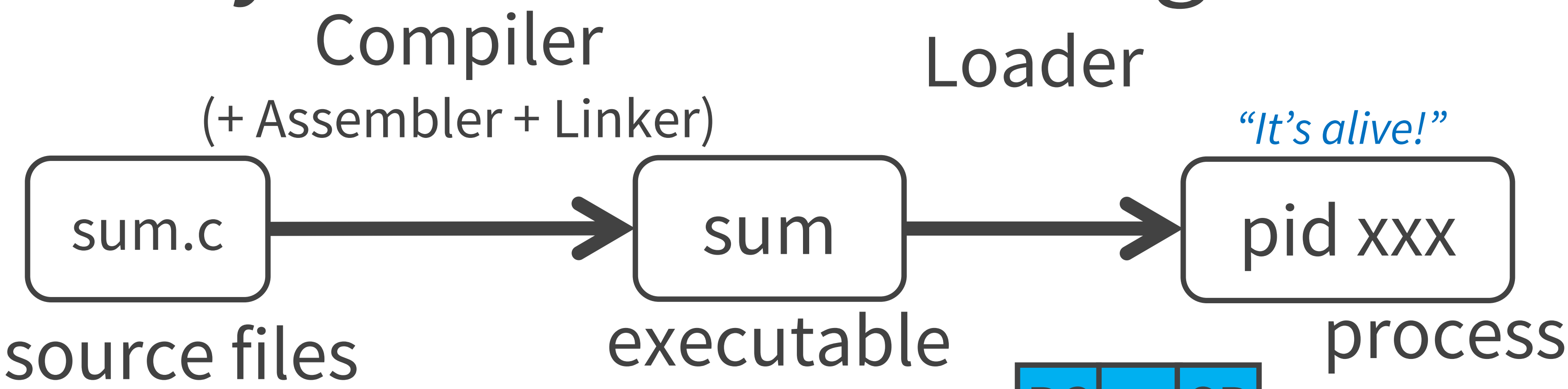
Virtualization

Want to create the illusion of more resources than exist in underlying physical system

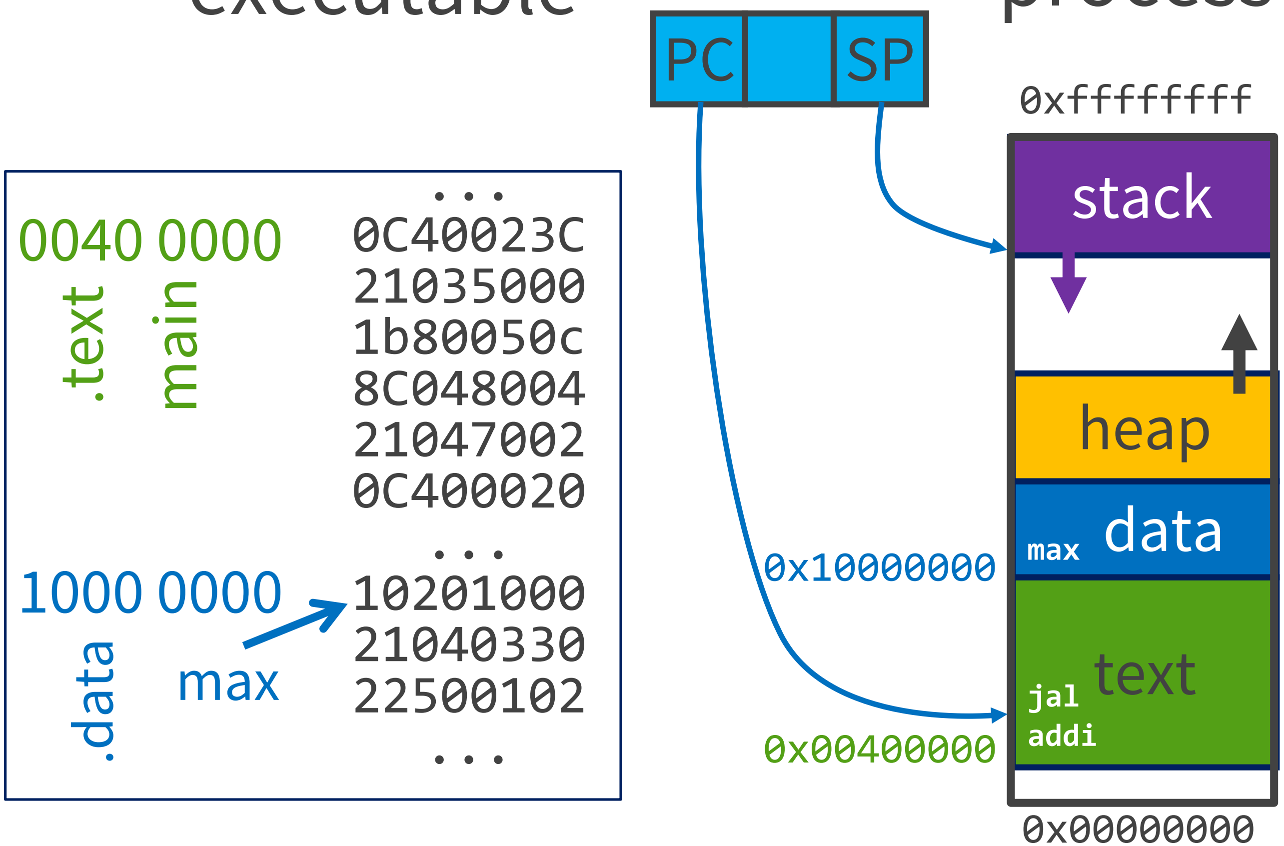
Utilization

Want to best use of this limited resource

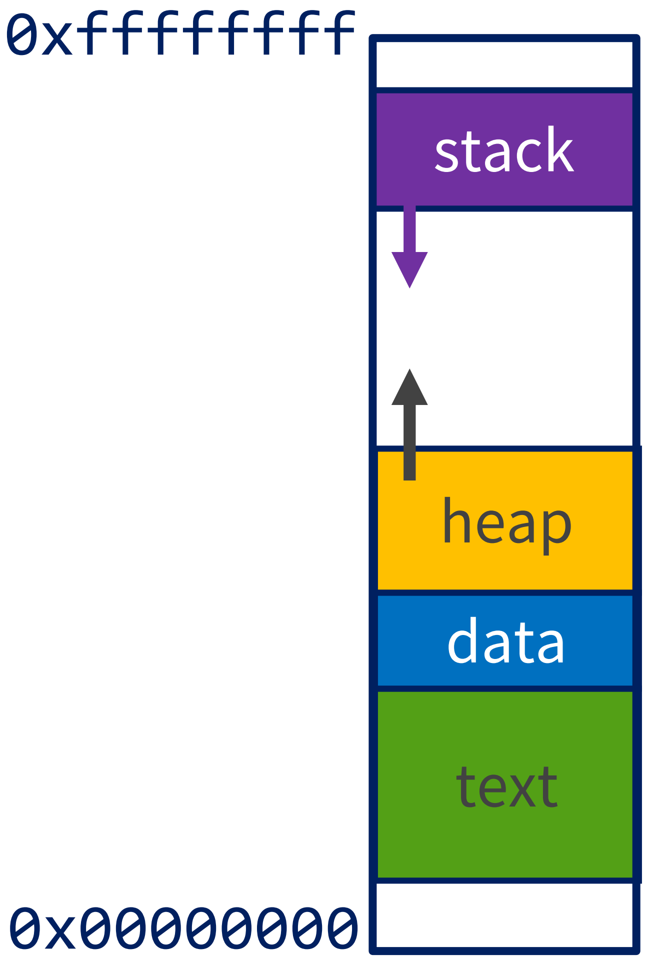
A Day in the Life of a Program




```c
#include <stdio.h>

int max = 10;

int main () {
    int i;
    int sum = 0;
    add(m, &sum);
    printf("%d",i);
    ...

}
```



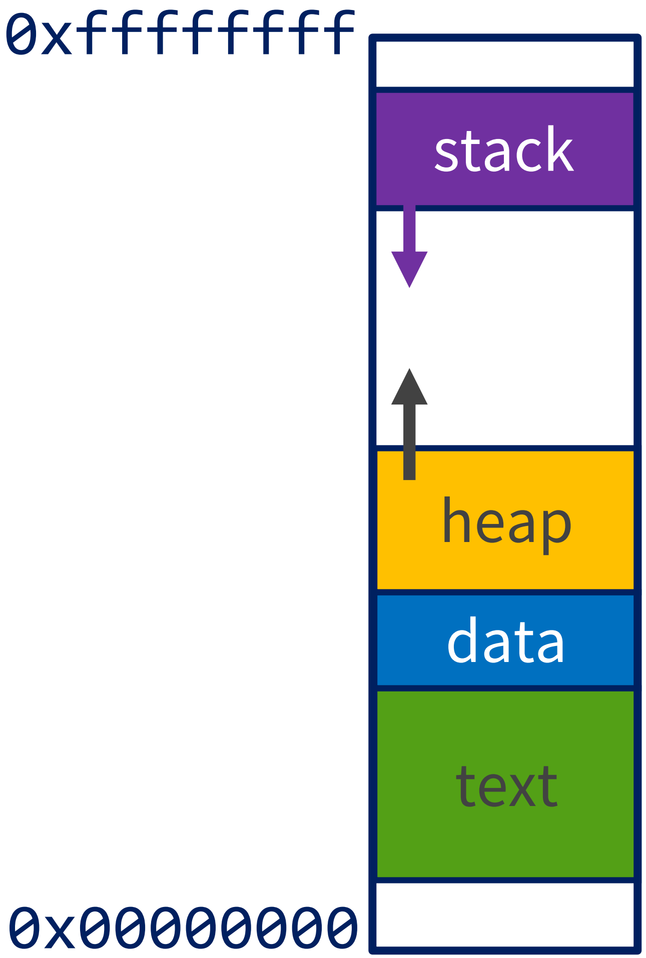
Logical view of process memory

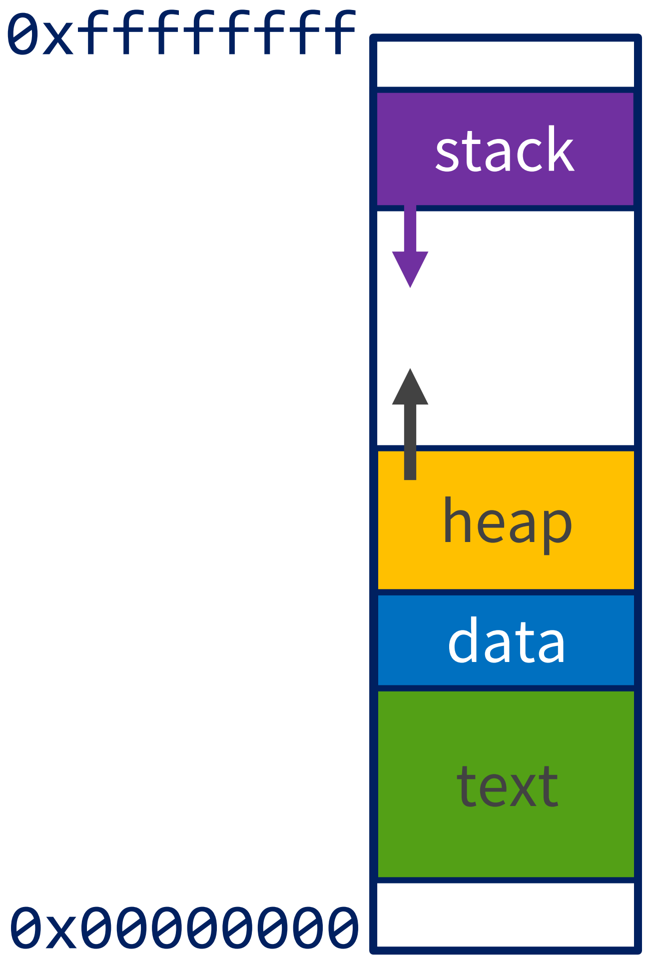


Where does this go in physical memory?

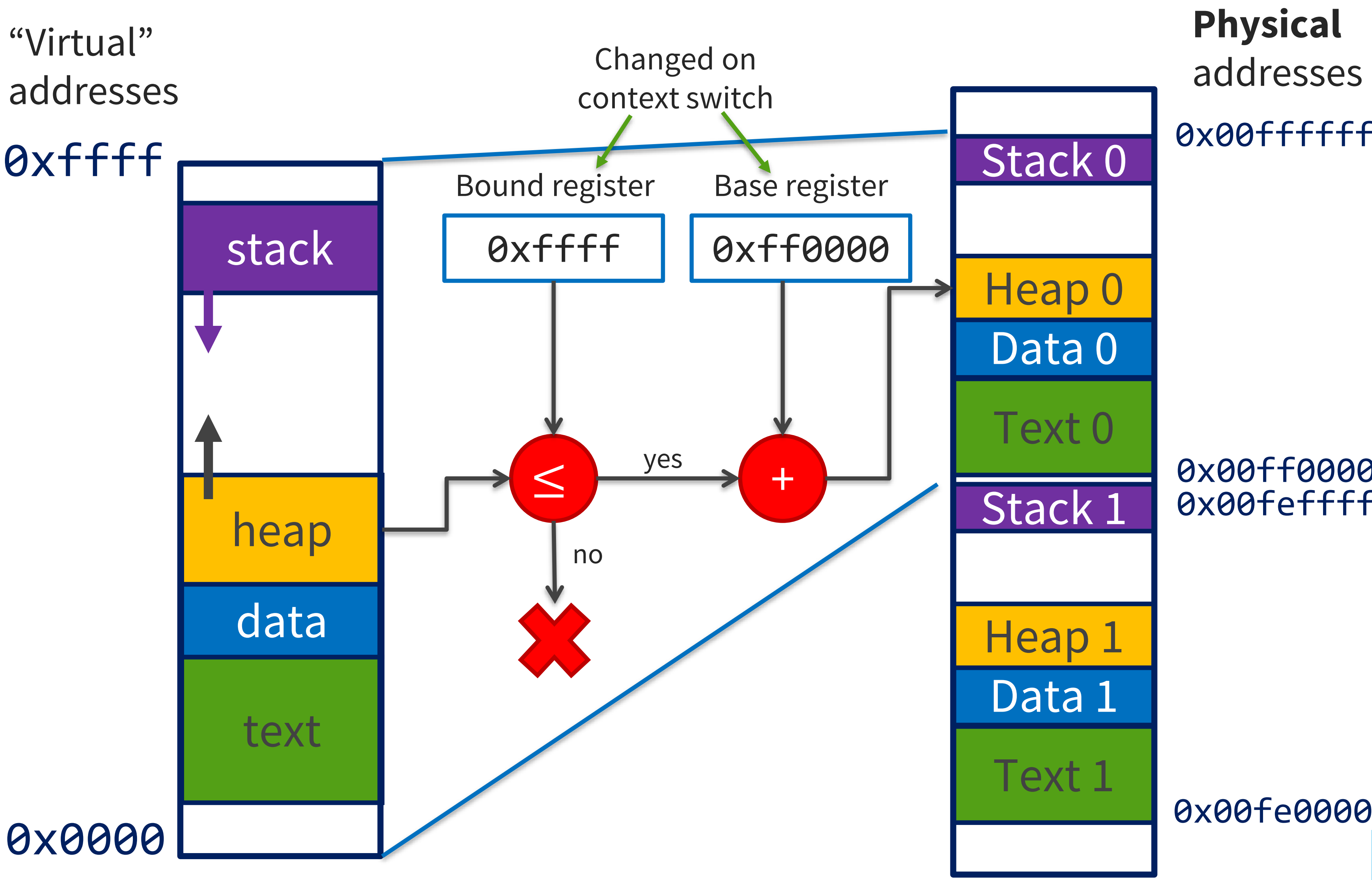
Logical view of process memory

What if we have 2 processes?



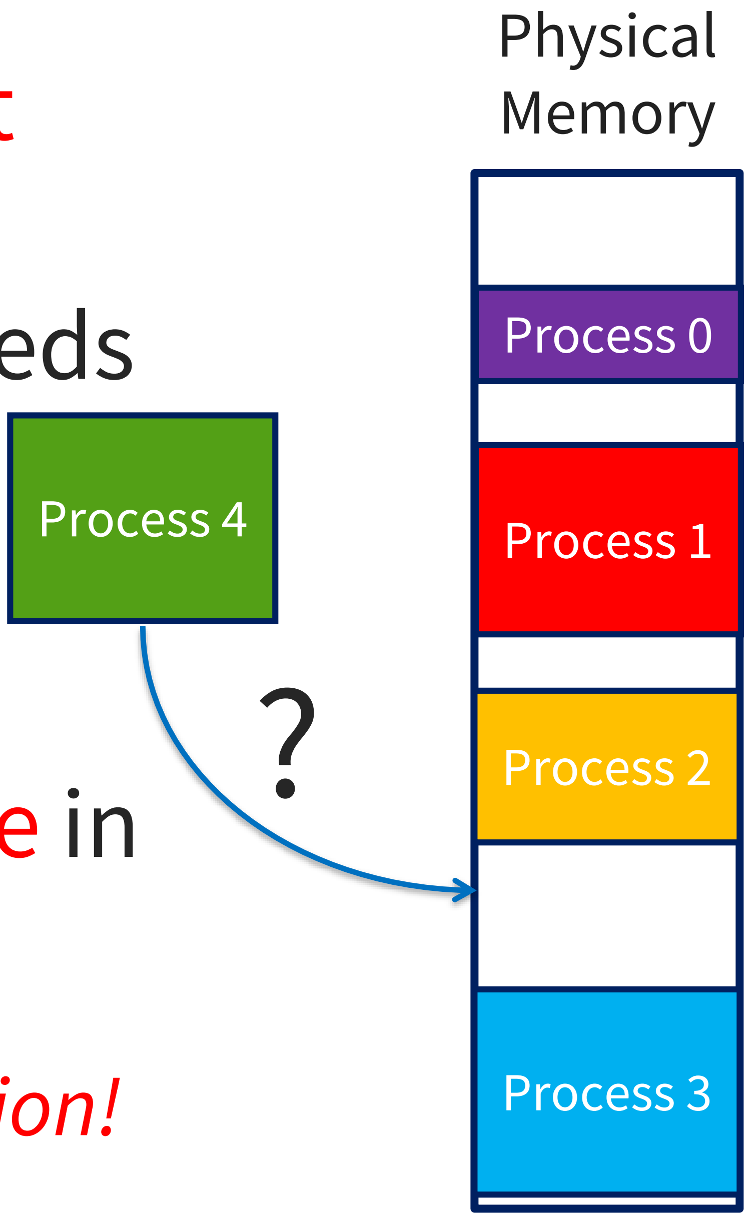


First attempt: Base + Bounds



Problems

- Processes use different amounts of memory
- Processes' memory needs change over time
- What happens when a new process can't fit into a contiguous space in physical memory?



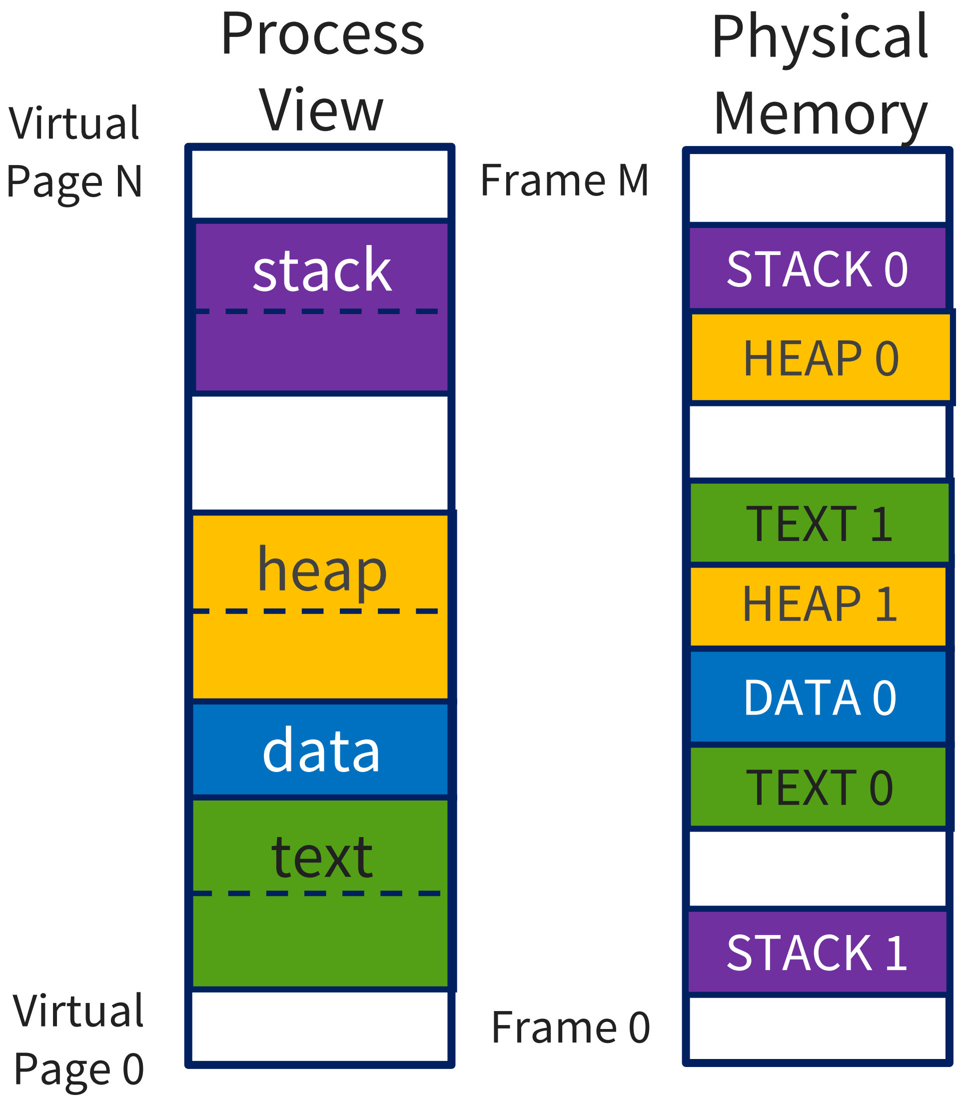
External fragmentation!

Paged Translation

TERMINOLOGY ALERT:

Page: the data itself

Frame: physical location



No more external fragmentation!

Paging Overview

Divide:

- Physical memory into fixed-sized blocks called **frames**
- Logical memory into blocks of same size called **pages**

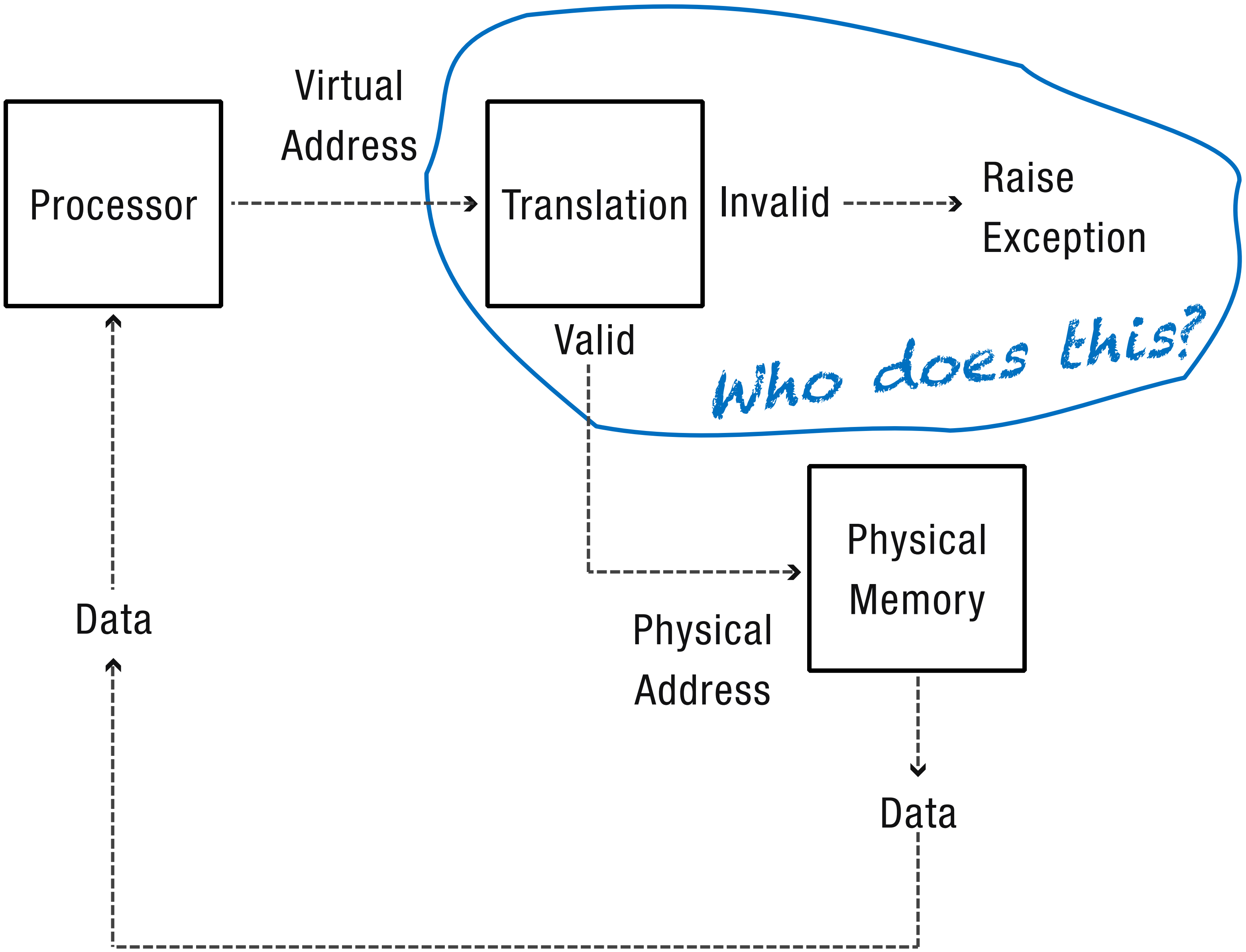
Management:

- Keep track of all free frames.
- To run a program with ***n*** pages, need to find ***n*** free frames and load program

Notice:

- Logical address space can be noncontiguous!
- Process given frames when/where available

Address Translation, Conceptually



Memory Management Unit (MMU)

- Hardware device
- Maps virtual to physical address (used to access data)

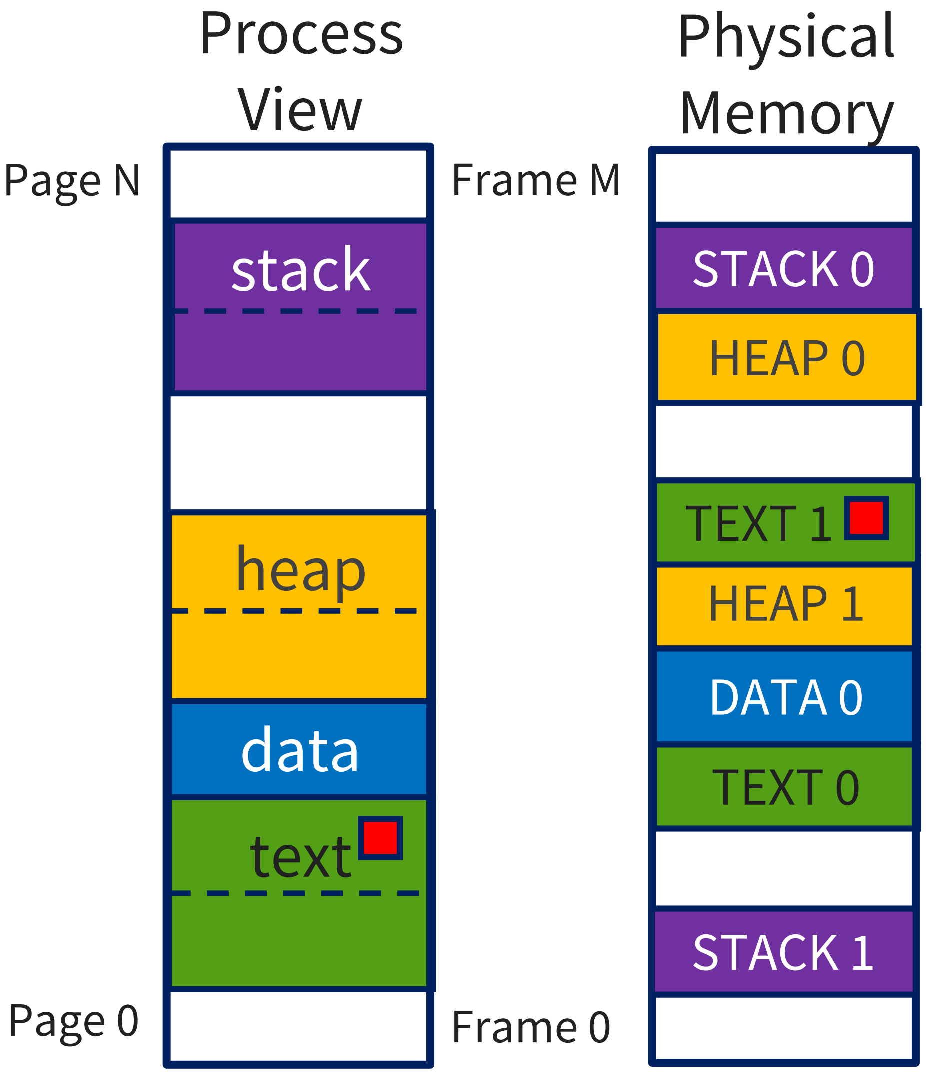
User Process:

- deals with *virtual* addresses
- Never sees the physical address

Physical Memory:

- deals with *physical* addresses
- Never sees the virtual address

High-Level Address Translation



red cube is 255th byte in page 2.

Where is the red cube in physical memory?

Logical Address Components

Page number – Upper bits

- Must be translated into a physical frame number

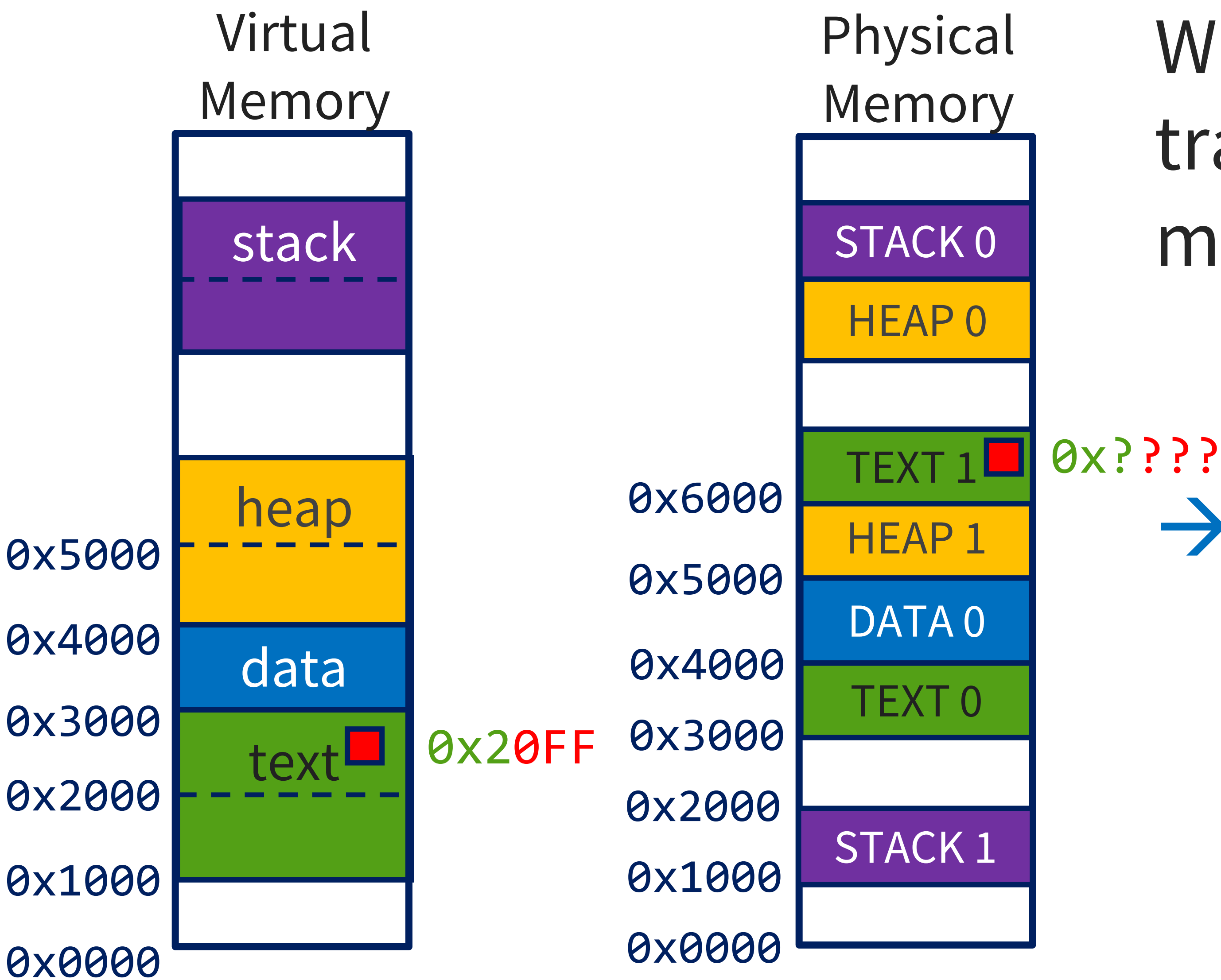
Page offset – Lower bits

- Does not change in translation

page number	page offset
m - n	*n*

For given logical address space 2^m and page size 2^n

High-Level Address Translation

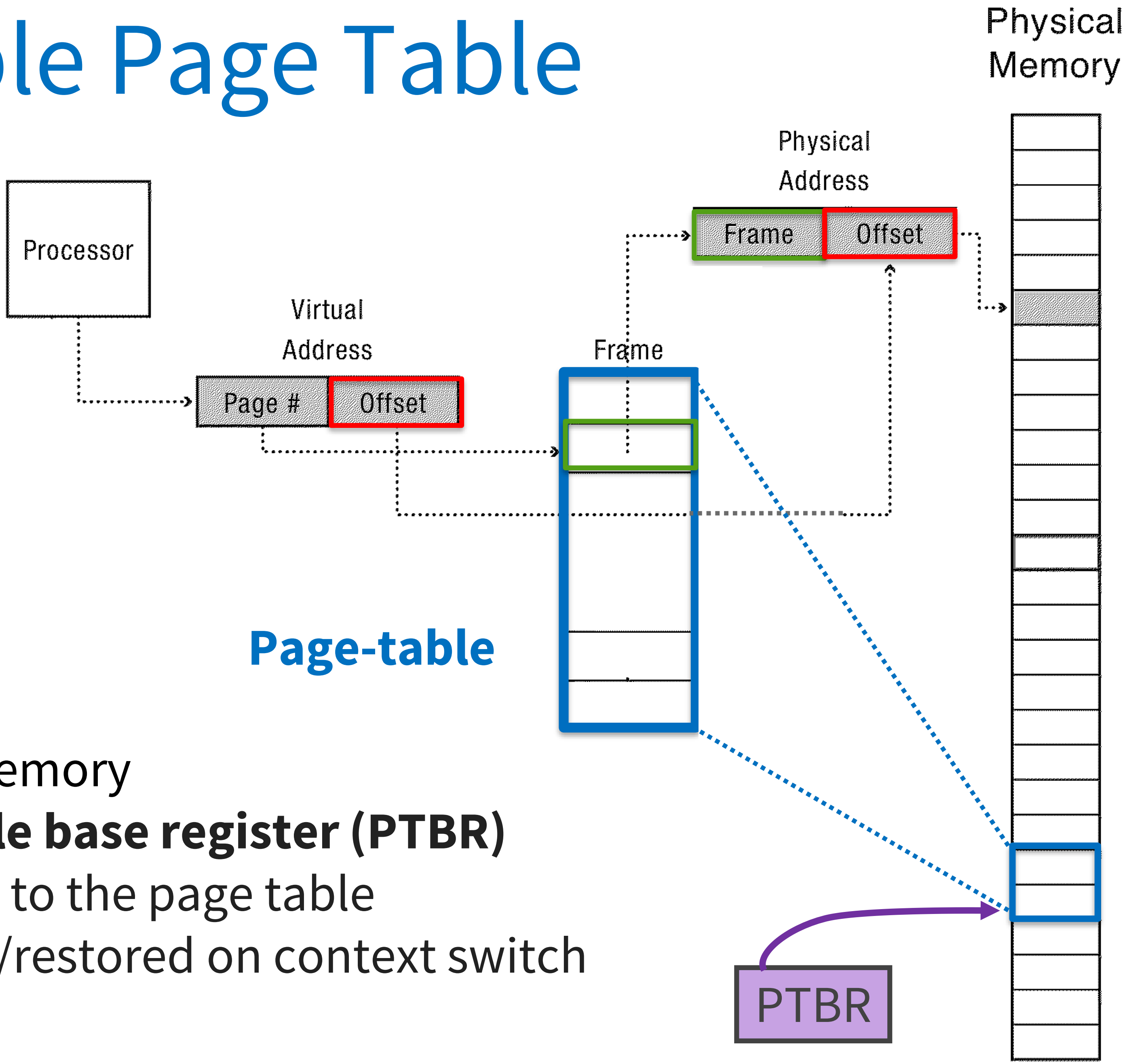


Who keeps track of the mapping?

→ Page Table

0	-
1	3
2	6
3	4
4	8
5…	5

Simple Page Table



Lives in Memory

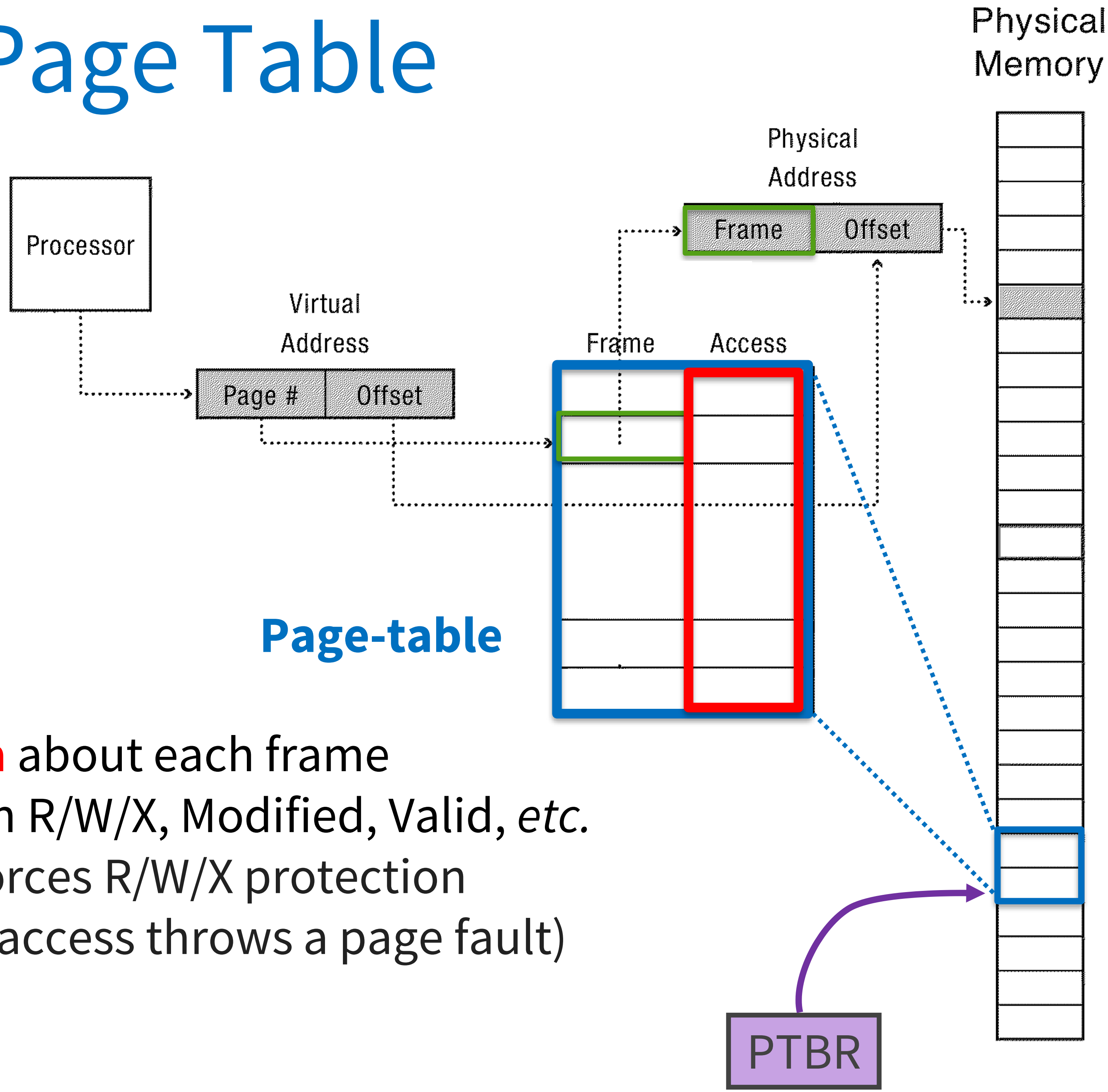
Page-table base register (PTBR)

- Points to the page table
- Saved/restored on context switch

Leveraging Paging

- Protection
- Dynamic Loading
- Dynamic Linking
- Copy-On-Write

Full Page Table



Meta Data about each frame

Protection R/W/X, Modified, Valid, *etc.*

MMU Enforces R/W/X protection

(illegal access throws a page fault)

Leveraging Paging

- Protection
- Dynamic Loading
- Dynamic Linking
- Copy-On-Write

Dynamic Loading & Linking

Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- No special support from the OS needed

Dynamic Linking

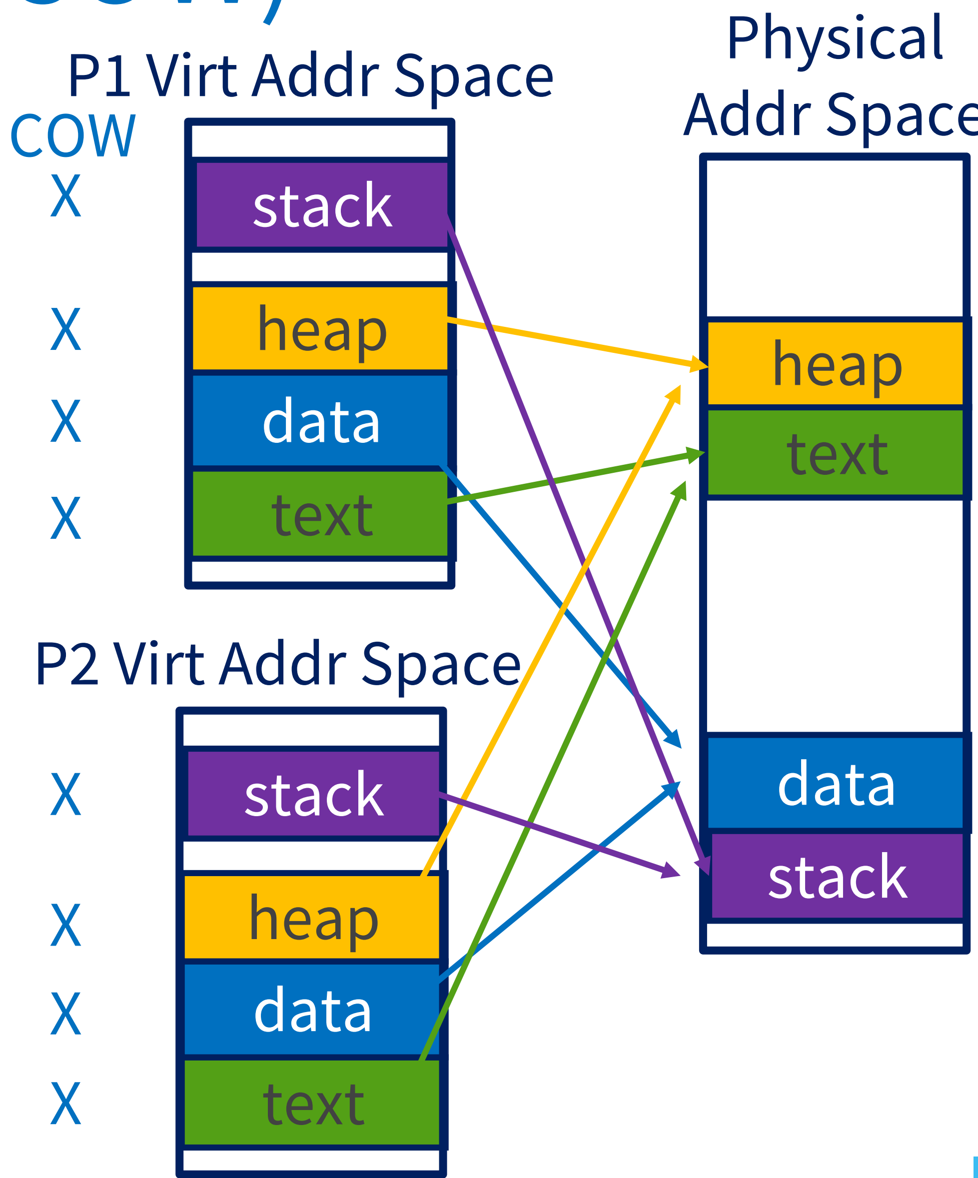
- Routine is not linked until execution time
- Locate (or load) library routine when called
- AKA **shared libraries** (*e.g.*, DLLs)

Leveraging Paging

- Protection
- Dynamic Loading
- Dynamic Linking
- Copy-On-Write

Copy on Write (COW)

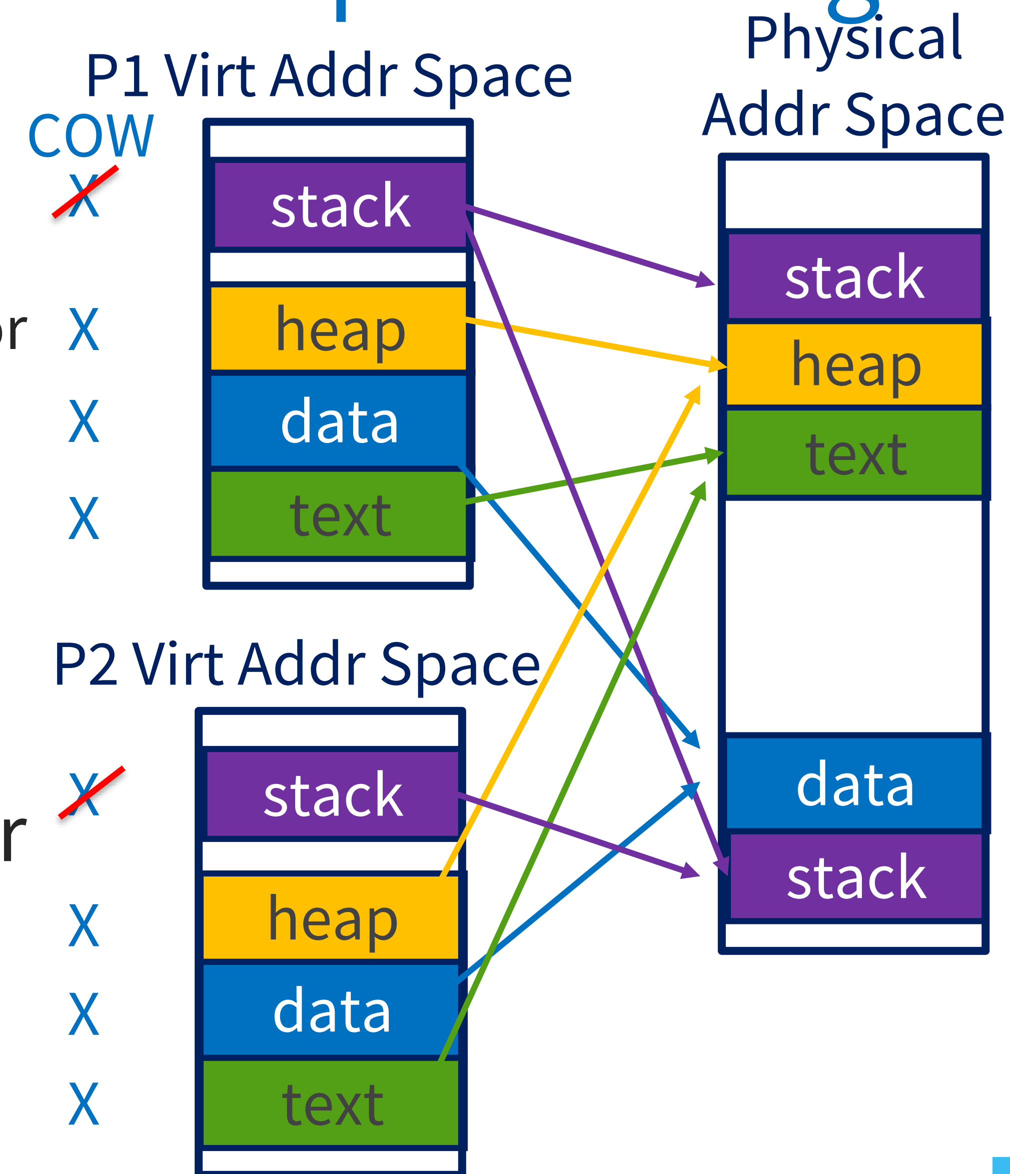
- P1 forks()
- P2 created with
  - own page table
  - same translations
- All pages marked COW (in Page Table)



Option 1: fork, then keep executing

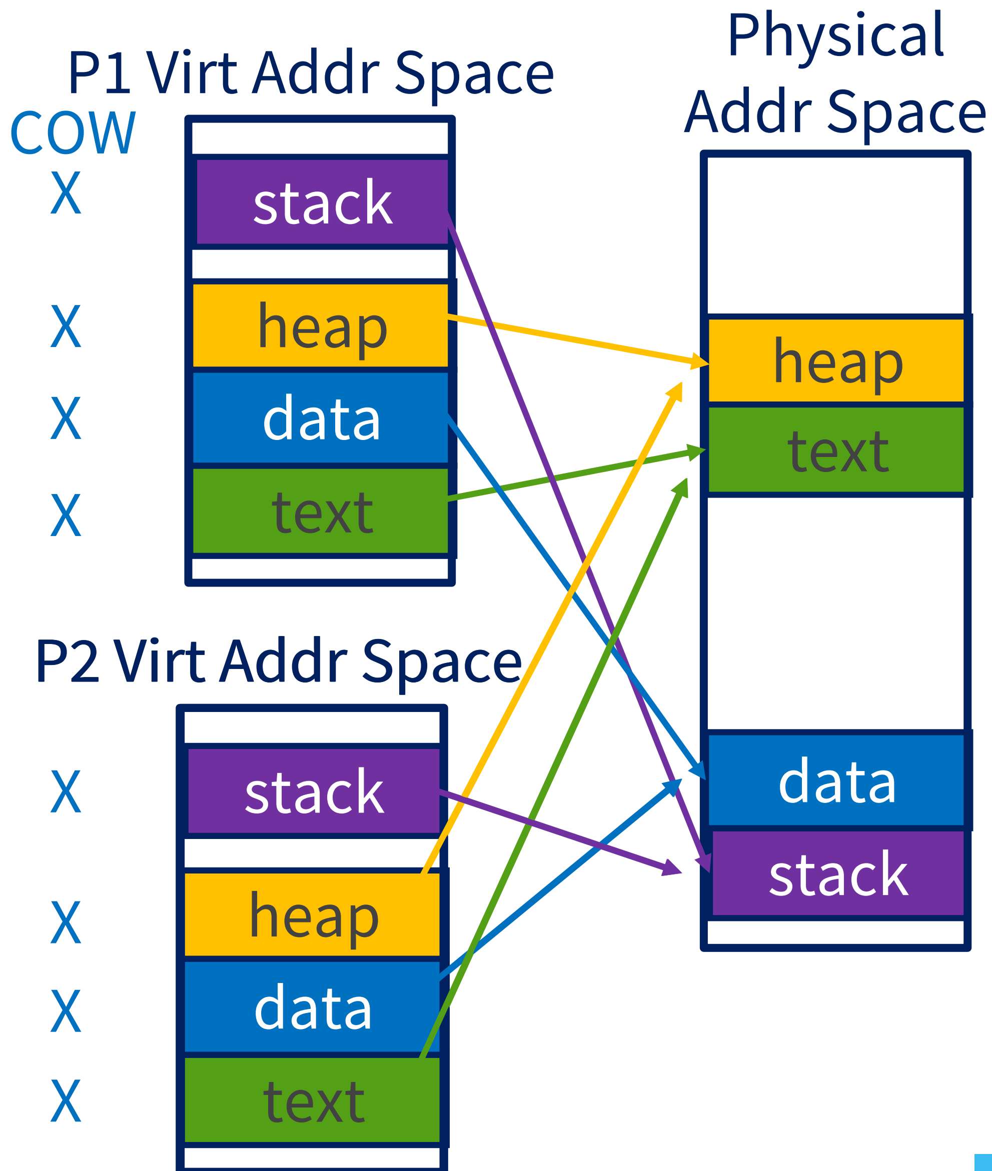
Now one process tries to write to the stack (for example):

- Page fault
- Allocate new frame
- Copy page
- Both pages no longer COW



Option 2: fork, then call exec

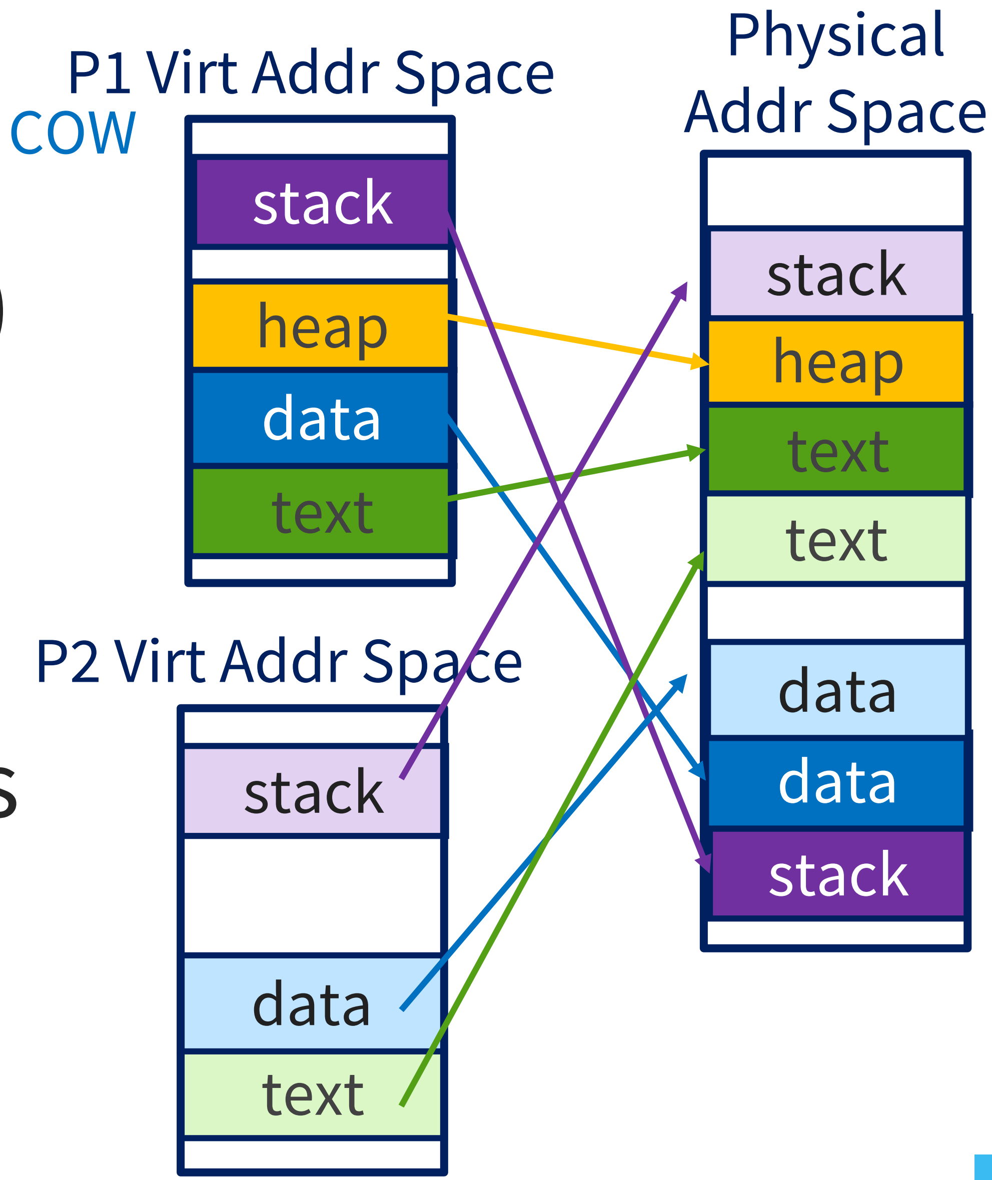
Before P2 calls exec()



Option 2: fork, then call exec

After P2 calls exec()

- Allocate new frames
- Load in new pages
- Pages no longer COW



Downsides to Paging

Memory Consumption:

- Internal Fragmentation
  - Make pages smaller? But then…
- Page Table Space: consider 32-bit address space, 4KB page size, each PTE 8 bytes
  - How big is this page table?
  - How many pages in memory does it need?

Performance: every data/instruction access requires *two* memory accesses:

- One for the page table
- One for the data/instruction